Rémi Pas – D2SI/SIDT

Version 1.0

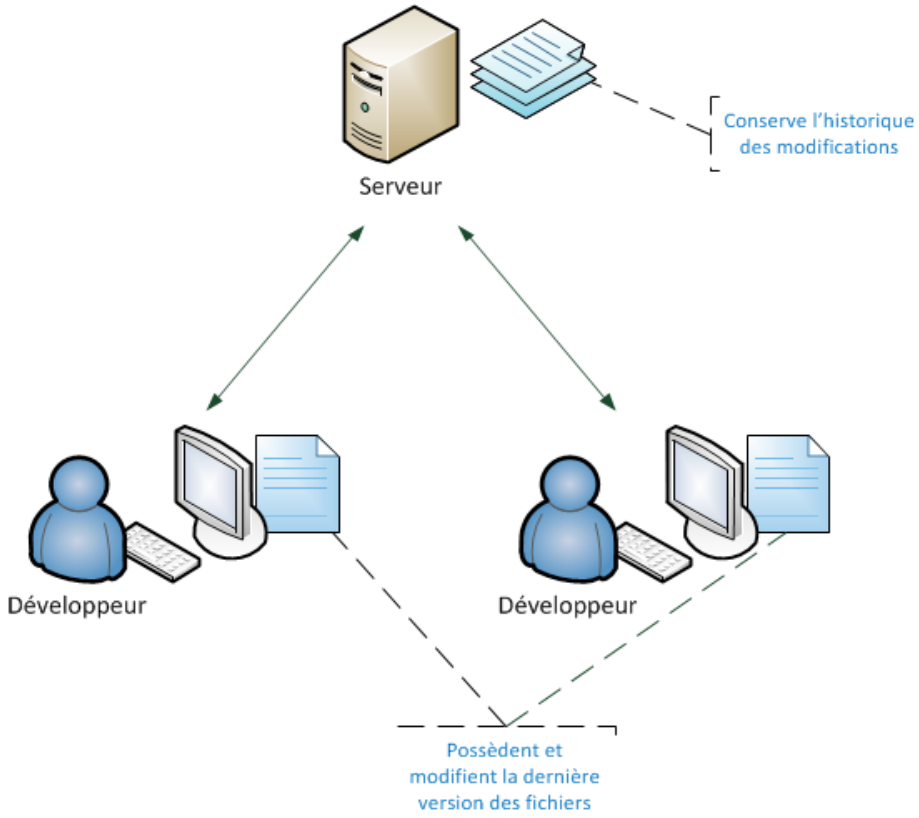
Avril 2015

Table des matières

1- Qu'est-ce que Git ?

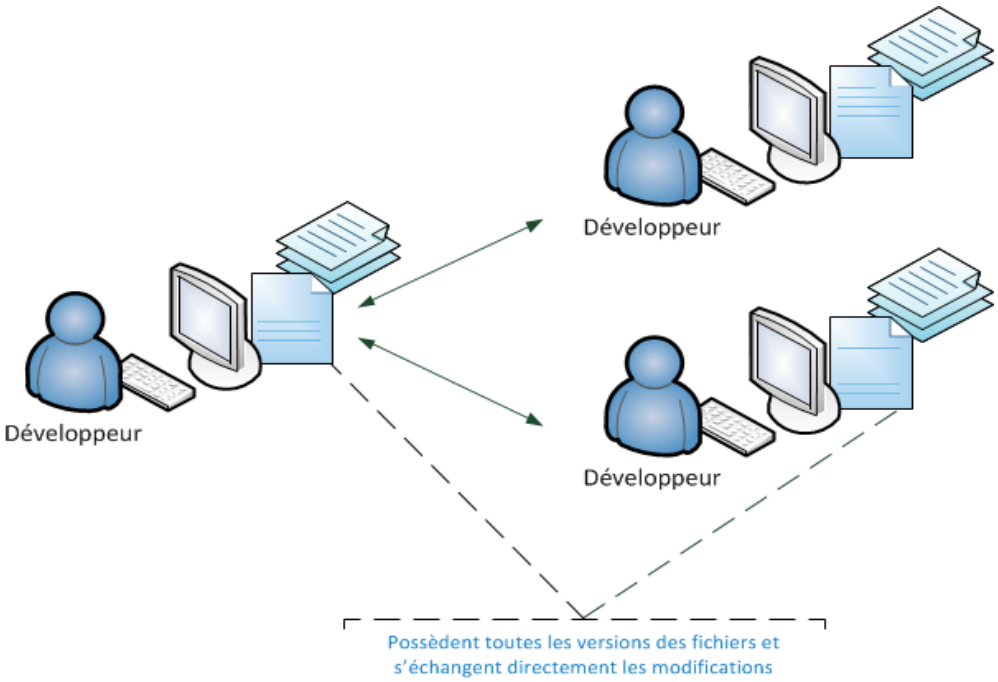
Git est un logiciel de gestion de version. Il a été développé initialement par Linus Torvalds à partir de 2005 afin de remplacer BitKeeper, le logiciel de gestion de version propriétaire utilisé pour le noyau Linux.

Contrairement à Subversion, Git est un système décentralisé : chaque développeur possède sur sa machine tout l'historique du projet. Lorsque l'on vient du monde Subversion, la première chose à faire pour apprendre à utiliser Git est donc de tout oublier.



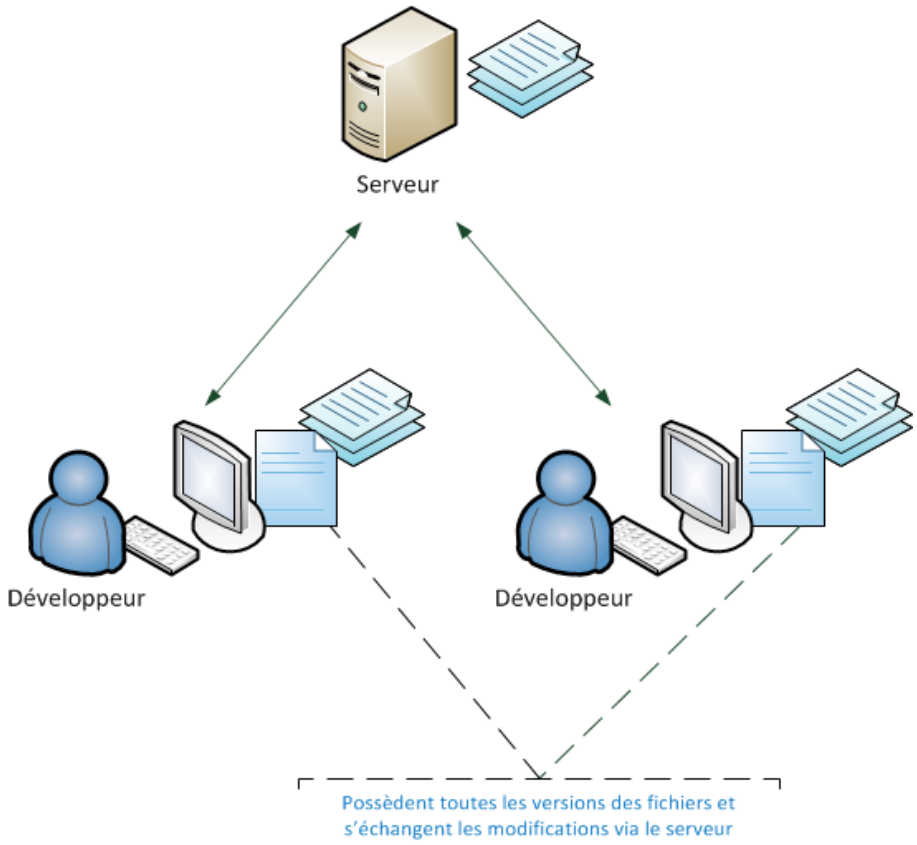
Fonctionnement d'un système centralisé tel que Subversion (cf. [NEB])

Sur un système centralisé comme le montre la figure précédente, le serveur centralise tout, et chaque développeur communique avec le serveur pour récupérer ou pousser du code. Sur son poste de travail, le développeur ne possède qu'une version du fichier source.



Fonctionnement d'un système décentralisé tel que Git (cf. [NEB])

Sur un système décentralisé, chaque développeur possède toutes les versions de chaque fichier. Les développeurs peuvent communiquer directement entre eux. Néanmoins, dans la pratique, il est également possible de posséder un (ou plusieurs) serveur(s) qui sert de copie distante du dépôt local. Les développeurs se servent alors de ce dépôt distant pour communiquer entre eux (ce point sera détaillé plus loin).



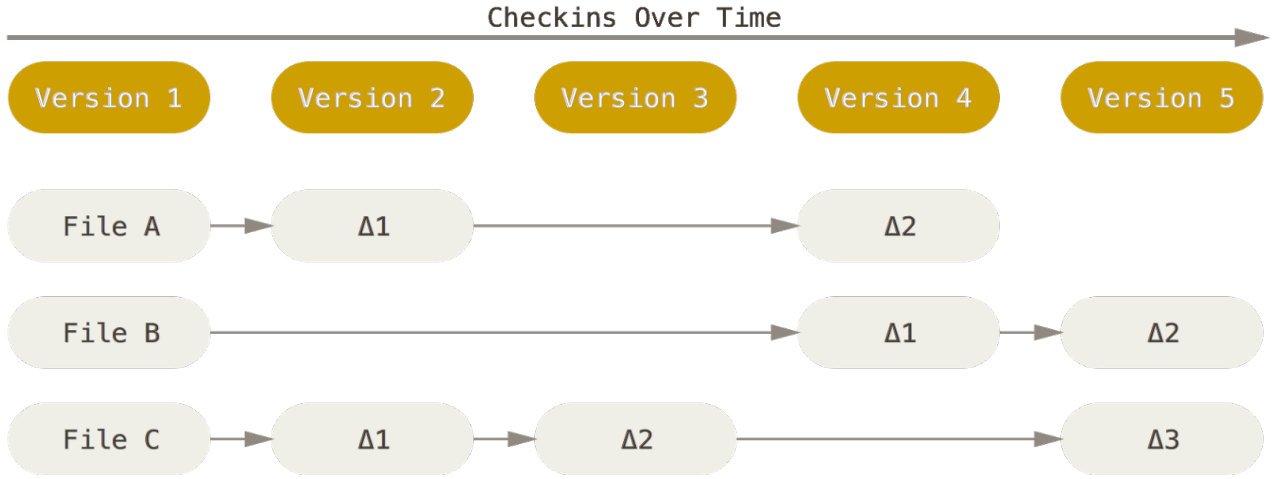
Git permet d'avoir une copie du dépôt local sur un serveur pour faciliter la communication entre les développeurs (cf. [NEB])

Avant d’aller plus loin, ce qu’il faut retenir de Git, c’est que **tout est local !** La preuve, voici le bandeau du site officiel de Git (http://git-scm.com) :

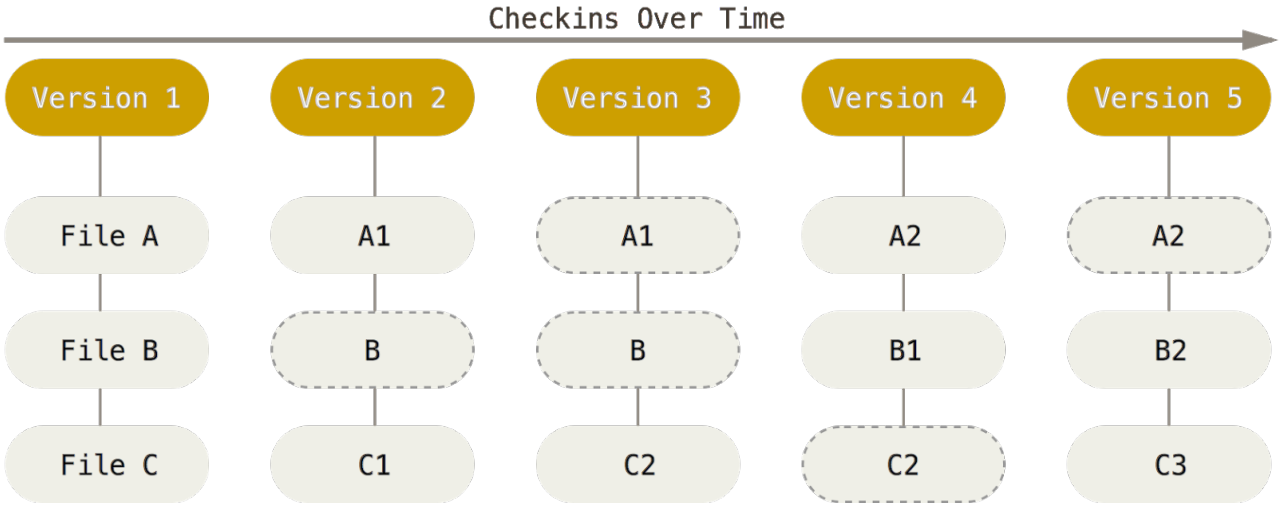
2- Un peu de théorie

2.1- Différences avec les autres systèmes

Outre le fait que Git est décentralisé, la gestion des fichiers se fait aussi très différemment. Là où Subversion stocke des différentiels entre chaque version, Git stocke des “snapshots”, c’est-à-dire une photographie de l’état de chaque fichier à un instant précis. L’avantage est que la reconstruction d’un dépôt sur un commit précis est très rapide : il suffit d’aller chercher la bonne version d’un fichier plutôt que de le reconstruire intégralement. La fusion entre deux fichiers est aussi nettement plus efficace, et la gestion des conflits s’en trouve facilitée.



La gestion de l’historique par différentiels dans un SCM classique (cf. [GITREF])



La gestion de l’historique par snapshots dans Git (cf. [GITREF])

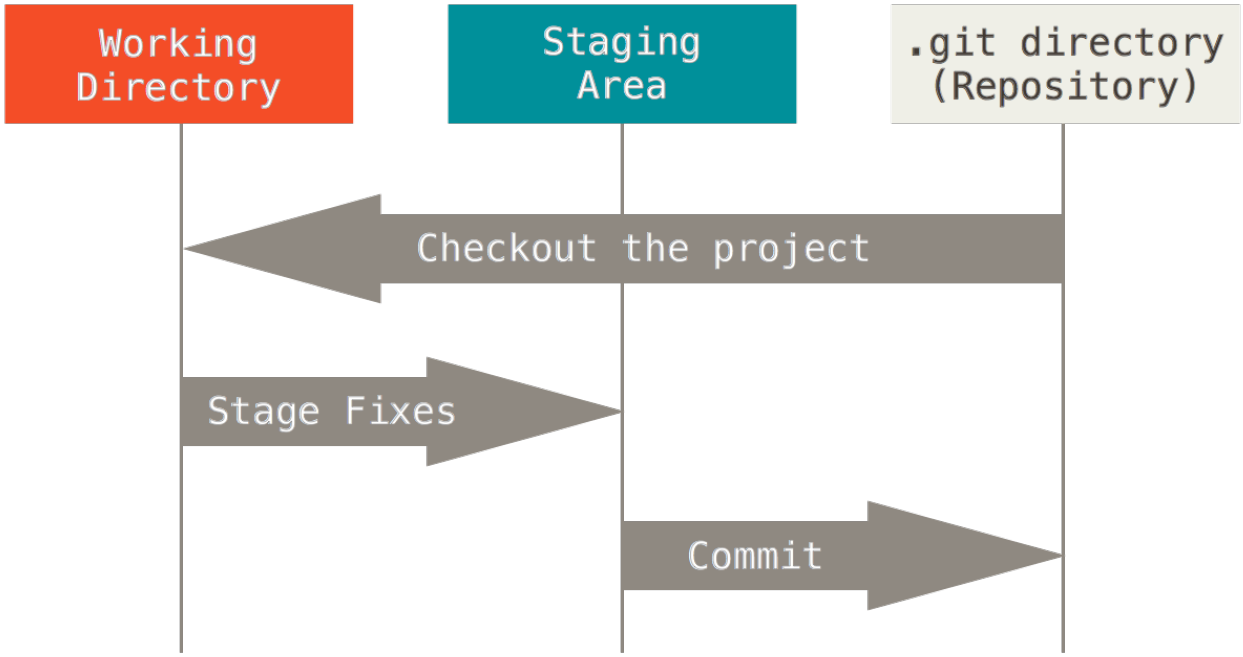
2.2- Les trois zones de Git

En vérité il y en a cinq. Mais les trois qui vont être présentées ici sont celles qui sont essentielles, les deux autres seront étudiées dans un second temps.

Le fonctionnement de ces trois zones constitue la notion principale à comprendre pour la maîtrise de Git. Une fois que ceci est acquis, le reste viendra tout seul.

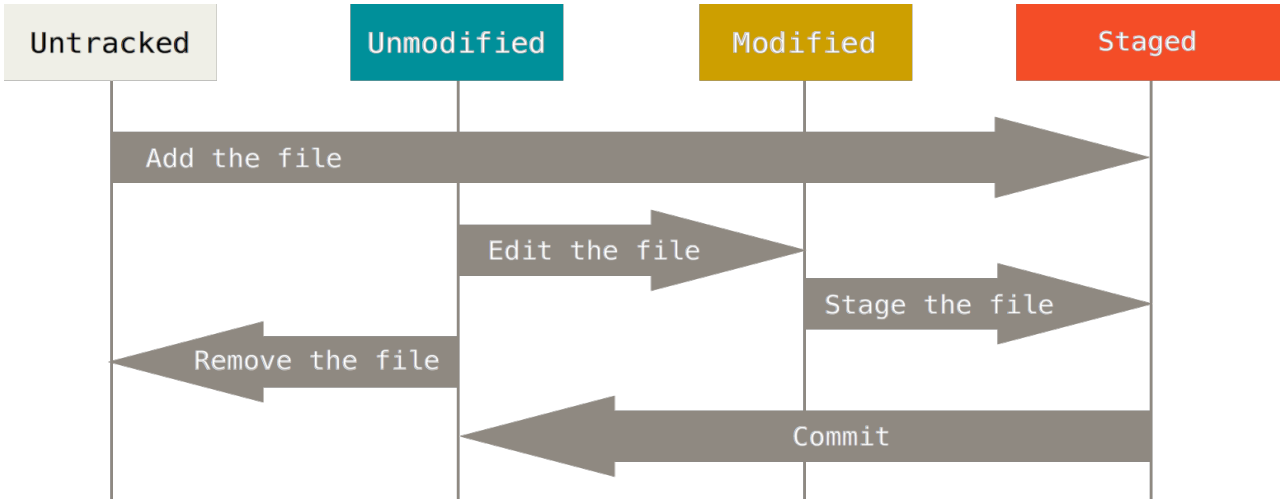
Ces trois zones sont :

- l'**espace de travail** (working directory) : il s'agit tout simplement du répertoire de travail, là où sont stockés les fichiers que le développeur manipule. Ces fichiers ont été extraits du dépôt à partir d'un commit particulier.
- l'**index** (staging area) : c'est une zone intermédiaire, dans laquelle on va stocker des informations sur ce que va contenir le prochain commit.
- le **dépôt** (repository) : il est matérialisé sous la forme d'un répertoire *.git*. C'est là que se trouve tout l'historique des fichiers (tous les snapshots).



Les trois principales zones de Git (cf. [GITREF])

Pour reprendre la formulation qui est faite dans [GITREF] : dans Git, un fichier peut posséder trois états : ***modified***, ***staged***, et ***committed (ou unmodified)***. Committed signifie que le fichier est stocké de manière sûre dans le dépôt. Modified signifie que quelque chose a changé dans le fichier et qu'il n'a pas encore été commité. Staged signifie qu'un fichier modifié a été marqué (snapshot) et qu'il fera partie du prochain commit.



Cycle de vie d'un fichier (cf. [GITREF])

En pratique, voici ce que cela peut donner (la commande *git status* permet de vérifier l'état des fichiers) :

```
$ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la
copie de travail)

    Modifié:      fichier1.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git
commit -a")
```

Git status indique ici qu'un fichier a été modifié. Il va donc falloir le mettre dans l'index pour préparer le prochain commit. Cela se fait avec la commande *git add*.

```
$ git add fichier1.txt
$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)

    modifié:      fichier1.txt
```

Désormais les modifications sont notées comme validées. Il ne reste plus qu'à commiter :

```
$ git commit -m "modifs sur fichier1"
[master f2ba8c1] modifs sur fichier1
1 file changed, 1 insertion(+)

$ git status
Sur la branche master
rien à valider, la copie de travail est propre
```

Le fichier est désormais commité sur la branche master, avec une clé de hashage pour identifier ce commit :

```
$ git log --pretty=oneline
f2ba8c1e58a0aa35cfff79fd1620df9f9336b62d modifs sur fichier1
af27f418c2fbfed143d9baaccb6b3e271c1edca8 first commit
```

En pratique, il est possible de “sauter” la partie indexation (elle est faite implicitement) par la commande suivante :

```
$ git commit -a -m "modifs sur fichier1"
```

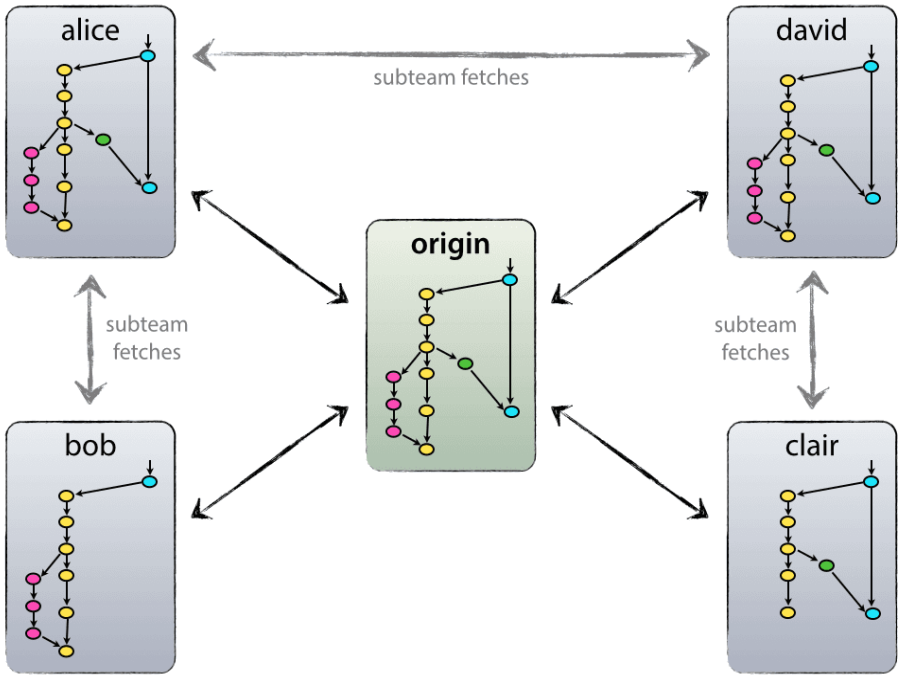
Le -a signifie add, il n’y a donc plus besoin de faire un *git add*. Ceci a pour effet de commiter tous les fichiers qui sont notés comme modifiés.

Les nouveaux fichiers s’ajoutent aussi dans l’index avec un *git add*.

Commande	**Effet**
`git status`	Montre l’état des fichiers (ainsi que la branche courante).
`git add`	Ajoute un fichier à l’index (ie : le prépare au prochain commit).
`git commit -m “message”`	Commite les fichiers de l’index avec un message.
`git commit -a -m “message”`	Commite directement les fichiers modifiés sans passer par l’étape de l’index.
`git log (--pretty=... --graph)`	Montre l’historique des commits (--pretty gère le format des lignes de l’historique, --graph affiche cela sous forme de graphe).

2.3- Une zone supplémentaire : le dépôt distant

Ce que l’on trouve sur le serveur distant est en fait une copie du dépôt local. Il n’y a pas d’espace de travail, ni d’index. Ce dépôt distant est simplement partagé par tous les développeurs. La notion de « distant » est une vue de l'esprit : c'est en fait un nouveau dépôt décentralisé. Un développeur peut tout à fait considérer le dépôt d'un autre développeur comme un dépôt distant.



Tout le monde peut être un dépôt distant (cf. [DRI])

On peut travailler avec plusieurs serveurs distants. Par défaut, le premier créé porte le nom de **origin**. Ce nom est un alias qui permet de ne pas réécrire à chaque l'adresse du serveur dans les commandes.

Les commandes principales pour interagir avec le serveur distant sont les suivantes :

Commande	**Effet**
`git clone <adresse-du-serveur>`	Copie le dépôt distant en local et met à jour l'espace de travail sur le dernier commit.
`git fetch origin <branche>`	Récupère les changements du dépôt distant sur une branche particulière et les copie dans le dépôt local, sans modifier l'espace de travail.
`git pull origin <branche>`	Récupère les changements du dépôt distant, les copie dans le dépôt local et modifie l'espace de travail avec ces modifications git pull = git fetch + git merge
`git push origin <branche>`	Envoie le contenu du dépôt local vers le dépôt distant. N'est possible que si le dépôt local n'est pas en retard sur le dépôt distant.

2.4- Zone bonus : la remise

C'est une zone très pratique qui permet d'envoyer des modifications qui ne sont pas encore commitées dans une zone tampon. Les modifications disparaissent de l'espace de travail, qui revient à son état

d’origine (celui du dernier commit), mais elles ne sont pas perdues, il est possible de les appliquer par la suite.

Par exemple :

```
$ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la
copie de travail)

    Modifié:      fichier2.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git
commit -a")
```

Des modifications ont été faites sur le fichier *fichier2.txt.* Pour une raison quelconque, on ne souhaite pas commiter tout de suite ces modifications (par exemple le travail n’est pas fini, mais on a besoin de faire autre chose en attendant).

```
$ git stash save "début de correction du bug #4"
Saved working directory and index state On master: début de correction du bug #4
HEAD est maintenant à 0dc8630 bug #3

$ git status
Sur la branche master
rien à valider, la copie de travail est propre

$ git stash list
stash@{0}: On master: début de correction du bug #4
```

git stash save a envoyé les modifications dans la remise, on ne les voie plus avec *git status.* Par contre *git stash list* montre bien que les modifications sont sauvées.

```
$ git stash apply
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la
copie de travail)

    Modifié:      fichier2.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git
commit -a")
```

git stash apply permet de récupérer une modification remisée.

Commande	**Effet**
`git stash`	Envoie les modifications dans la remise.

`git stash save <message>`	Même chose avec un message personnalisé.
`git stash apply <n° stash>`	Applique une modification remisée à l'espace de travail.
`git stash drop <n° stash>`	Supprime une modification de la remise.
`git stash pop`	Applique la dernière modification remisée à l'espace de travail et l'enlève de la remise.
`git stash list`	Liste toutes les modifications remisées.
`git stash show <n° stash>`	Montre les fichiers affectés par une modification.
`git stash branch <nom-branche> <n° stash>`	Crée une branche à partir d'un stash.
`git stash clear`	Vide la remise.

3- Fonctionnement au quotidien

3.1- Initialiser un dépôt

Commande	Effet
`git init`	Initialise un dépôt local. A exécuter dans le répertoire de son code source.
`git init --bare`	Initialise un dépôt distant.
`git remote add origin <adresse dépôt distant>`	Enregistre un dépôt distant nommé origin pour le dépôt local.

3.2- Cloner un dépôt distant

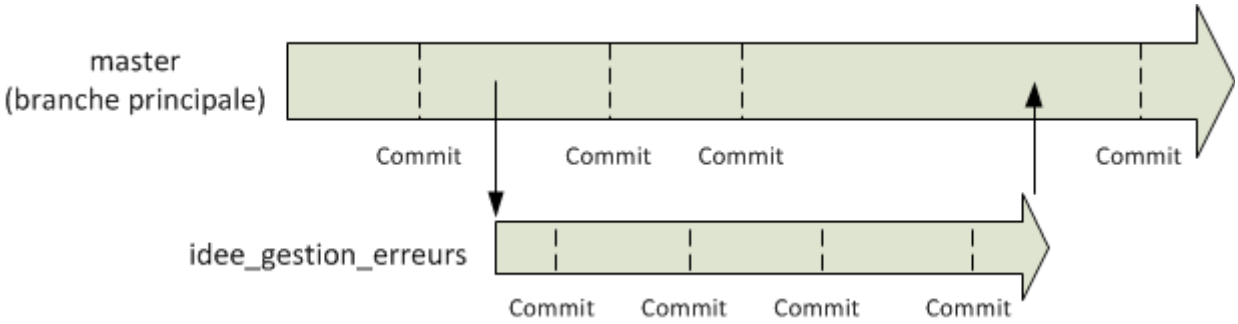
Lorsque le dépôt distant existe déjà, où que l'on travaille avec un système tel que Gitlab ou Github, on n'a pas besoin d'initialiser ce dépôt, mais plutôt de le récupérer.

Commande	Effet
`git clone <adresse dépôt distant>`	Récupère un dépôt distant et met le snapshot du dernier commit dans l'espace de travail.

3.3- Créer une branche et naviguer entre les branches

Le véritable intérêt de Git réside dans sa gestion des branches très souple. Contrairement à Subversion, où une branche est forcément sur le serveur et tout le monde la voit, sur Git tout est local : il est donc possible de faire une branche locale sur laquelle on va travailler une fonctionnalité particulière de manière isolée, puis une fois le travail terminé, fusionner cette branche avec la branche principale.

Par défaut, la branche principale s'appelle master.



Création d'une branche et fusion dans la branche master (cf. [NEB])

En pratique :

```
$ git branch idee_gestion_erreur
$ git checkout idee_gestion_erreur
Basculement sur la branche 'idee_gestion_erreur'
```

A ce stade, on se situe sur la branche idee_gestion_erreur, on peut commencer à travailler dessus.

```
$ git status
Sur la branche idee_gestion_erreur
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les modifications dans la
copie de travail)
    modifié:      fichier1.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git
commit -a")

$ git commit -a -m "gestion des erreurs"
[idee_gestion_erreur 146ec44] gestion des erreurs
1 file changed, 1 insertion(+), 1 deletion(-)
```

Pendant ce temps, du travail a pu également être réalisé sur la branche master, cela n'est pas gênant pour fusionner (il peut y avoir éventuellement un conflit à régler, mais cela reste rare).

```
$ git checkout master
$ git merge idee_gestion_erreur
$ git log --graph --pretty=oneline --abbrev-commit
*   57c95ef Merge branch 'idee_gestion_erreur'
|\
| * 146ec44 gestion des erreurs
* | 648be0d ajout de fichier2
|/
* ea88060 first commit
```

On voit alors apparaître un graphe en forme de bosse qui montre qu'une branche a existé et qui a été fusionnée.

Commande	Effet
`git branch`	Liste les branches, et montre celle sur laquelle on se situe.
`git branch <branche>`	Crée une branche.
`git checkout <branche>`	Place l'espace de travail sur une branche.
`git checkout -b <branche>`	Crée une branche et se place directement dessus.
`git branch -d <branche>`	Supprime une branche.
`git merge <branche>`	Fusionne la branche désignée dans la branche courante.

3.4- Branche serveur

Parfois, une branche a besoin d'être partagée entre les développeurs. C'est le cas par exemple des branches de release.

```
$ git push origin idee_gestion_erreur
Counting objects: 5, done.
Writing objects: 100% (3/3), 251 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/rpas/Devs/formation_git/remote/
* [new branch]    idee_gestion_erreur -> idee_gestion_erreur
```

Cette commande a créé une branche nommée *idee_gestion_serveur* sur le dépôt distant.

On peut ensuite récupérer la liste des branches distantes :

```
$ git fetch origin
$ git branch -a
  idee_gestion_erreur
* master
  remotes/origin/idee_gestion_erreur
  remotes/origin/master
  remotes/origin/v1.0
```

On voit ici qu'il existe une branche distante nommée v1.0. Si on souhaite travailler dessus, il faut alors créer sa version locale :

```
$ git checkout -b v1.0 origin/v1.0
La branche v1.0 est paramétrée pour suivre la branche distante v1.0 depuis origin.
Basculement sur la nouvelle branche 'v1.0'
```

On aurait également pu faire la commande précédente ainsi :

```
$ git checkout --track origin/v1.0
```

Commande	Effet
`git branch -a`	Liste toutes les branches, y compris celles du serveur.
`git push origin <branche>`	Envoie les modifications locale sur une branche serveur. Si la branche n'existe pas, elle est créée.
`git fetch origin`	Récupère les branches distantes (en plus de mettre à jour le dépôt).

`Git checkout -b <branche> origin/<branche>`
 	
`git checkout --track origin/<branche>`
	Crée une branche locale qui suit une branche distante.

`git push origin :<branche>`	Supprime la branche distante. **Dangereux, car facile de faire une faute de frappe.**

3.5- Tags

Git permet, comme tout bon SCM, de taguer certaines versions. Il s'agit en fait d'étiqueter un commit particulier, afin de le retrouver plus facilement.

Il y a trois sortes de tag avec Git :

- les tags légers : ils ne représentent qu'un alias d'un commit particulier.
- les tags annotés : ils représentent un véritable objet au sens de Git. Ils pointent sur un commit particulier, mais possèdent également certaines métadonnées (somme de contrôle, auteur, commentaire).
- les tags signés : ils sont semblables aux tags annotés, mais sont en plus signés avec GPG.

La création d'un tag se fait avec la commande *git tag* :

```
$ git tag -a v1.0.2 -m "Version 1.0.2"

$ git tag
v1.0.0
v1.0.1
v1.0.2
```

Pour se placer sur un tag particulier, il suffit de faire :

```
$ git checkout v1.0.1
Note: checking out 'v1.0.1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD est maintenant sur 7937877... commit pour version 1.0.1

$ git status
HEAD détachée sur v1.0.1
rien à valider, la copie de travail est propre
```

Les tags peuvent être envoyés sur le serveur :

```
$ git push origin v1.0.0
Counting objects: 1, done.
Writing objects: 100% (1/1), 155 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To /home/rpas/Devs/formation_git/remote/
* [new tag]       v1.0.0 -> v1.0.0
```

Ou plus simplement, pour envoyer tous les tags en une seule opération :

```
$ git push origin --tags
Counting objects: 2, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 279 bytes | 0 bytes/s, done.
Total 2 (delta 0), reused 0 (delta 0)
To /home/rpas/Devs/formation_git/remote/
* [new tag]       v1.0.1 -> v1.0.1
* [new tag]       v1.0.2 -> v1.0.2
```

Commande	**Effet**
`git tag`	Liste les tags existants.
`git tag <tag>`	Crée un tag léger.

`git tag -a <tag> -m <message>`	Crée un tag annoté.
`git tag -s <tag> -m <message>`	Crée un tag annoté et signé.
`git show <tag>`	Montre les informations d'un tag (pour les tags annotés).
`git checkout <tag>`	Se détache sur un tag.
`git push origin <tag>` `git push origin --tags`	Envoie les tags sur le serveur.

4- Git avancé

4.1- Cherry-pick

Les branches de Git sont très utiles pour développer des fonctionnalités ou faire des corrections de manière isolée. Mais imaginons le cas suivant : un projet contenant deux branches, l'une master sur laquelle on fait le développement[1], et l'autre 1.0 qui sert au logiciel déployé en production et sur laquelle on fait la maintenance. En théorie, tout opération de maintenance sur la branche 1.0 doit être mergée sur master afin que cette correction soit présente sur les futures branches de release.

Un développeur étourdi fait sa correction sur la branche master au lieu de 1.0. Il est évidemment hors de question de refaire une seconde fois le travail dans la branche 1.0 et encore moins de fusionner master dans 1.0 !

La solution dans ce cas est de faire un cherry-pick, c'est-à-dire appliquer un commit particulier à la branche sur laquelle on se trouve.

```
$ git log --graph --pretty=oneline --abbrev-commit
* 190c51f fixes bug #5
* 8a1e34a ajout d'un bouton qui fait tout
*   57c95ef Merge branch 'idee_gestion_erreur'
|\
| * 146ec44 gestion des erreurs
* | 648be0d ajout de fichier2
|/
* ea88060 first commit
```

Voici le graphe de la branche master. Le commit *190c51f* corrige un bug qui devait être fait sur la branche 1.0.

```
$ git checkout v1.0
Basculement sur la branche 'v1.0'
Votre branche est à jour avec 'origin/v1.0'.

$ git cherry-pick 190c51f
[v1.0 a39d540] fixes bug #5
 1 file changed, 1 insertion(+)
 create mode 100644 fichier3.txt

$ git log --graph --pretty=oneline --abbrev-commit
* a39d540 fixes bug #5
* f765138 commit pour version 1.0.2
* 7937877 commit pour version 1.0.1
* 6712b4f sortie v1.0
* ea88060 first commit
```

Le cherry-pick a permis de ramener la correction faite sur master dans la branche 1.0.

1 Ce qui n'est pas forcément une bonne pratique,

Commande	Effet
`git cherry-pick <commit>`	Applique le commit désigné sur la branche courante.

4.2- Merge et rebase

La notion de merge a déjà été abordée précédemment : elle permet de fusionner deux branches ou plus spécifiquement d'appliquer les commits faits à une branche sur une autre.

Avec Git, on peut faire des branches pour deux raisons :

- Parce que l'on souhaite travailler de manière isolée sur l'implémentation d'une fonctionnalité sans perturber le reste du développement. Ces branches sont en général locales et non partagées.
- Parce que l'on souhaite créer une branche de release qui servira pour la production ou que l'on souhaite faire une branche spéciale pour une fonctionnalité particulière (ou expérimentale) que l'on veut partager avec le reste de l'équipe. Ces branches sont en général poussées sur le serveur.

Dans le premier cas, la branche a vocation à disparaître une fois la fonctionnalité implémentée. Au moment du merge, l'historique va présenter la forme d'une bosse, comme on peut le voir ici :

```
$ git log --graph --pretty=oneline --abbrev-commit --all
*   43d2178 Merge branch 'branch_1'
|\
| * 39cc76b fixes bug #2
| * f058f70 fixes bug #1
* | 7376799 feature #3
|/
* ad09996 feature #2
* 8fe2fc3 feature #1
* bc74038 first commit
```

Les autres développeurs voient donc une trace de la présence de cette branche, ce qui ajoute de la confusion inutile dans des historiques déjà difficiles à exploiter.

Avant de faire un merge, il convient donc de se poser une question : **faut-il que ma branche soit visible des autres ?** Plusieurs scénarios sont envisageables.

4.2.1- Je ne veux pas que ma branche soit visible

Git peut fusionner les branches de deux façons différentes :

- **avec un *fast-forward* :** c'est ce qu'il se produit lorsque la branche cible n'a pas bougé depuis la création de la branche. Dans ce cas, la branche source n'apparaîtra pas dans l'historique.

```
$ git checkout -b "branch_1"
Basculement sur la nouvelle branche 'branch_1'
…
$ git commit -a -m "fixes bug #1"
…
$ git commit -a -m "fixes bug #2"

$ git checkout master
Basculement sur la branche 'master'

$ git merge branch_1
Mise à jour ad09996..614dfd4
Fast-forward
 fichier1.txt | 2 ++
 1 file changed, 2 insertions(+)

$ git log --graph --pretty=oneline --abbrev-commit
* 614dfd4 fixes bug #2
* 03d80b7 fixes bug #1
* ad09996 feature #2
* 8fe2fc3 feature #1
* bc74038 first commit
```

- **en faisant un *true merge*** (c'est-à-dire sans fast-forward) : c'est le comportement par défaut quand la branche cible a bougé depuis la création de la branche source.

Supposons le dépôt dans l'étant suivant :

```
$ git log --graph --pretty=oneline --abbrev-commit --all
* 7376799 feature #3
| * 39cc76b fixes bug #2
| * f058f70 fixes bug #1
|/
* ad09996 feature #2
* 8fe2fc3 feature #1
* bc74038 first commit
```

Un git merge produit ceci :

```
$ git merge branch_1
Merge made by the 'recursive' strategy.
 fichier1.txt | 2 ++
 1 file changed, 2 insertions(+)

$ git log --graph --pretty=oneline --abbrev-commit
*   43d2178 Merge branch 'branch_1'
|\
| * 39cc76b fixes bug #2
| * f058f70 fixes bug #1
* | 7376799 feature #3
|/
* ad09996 feature #2
* 8fe2fc3 feature #1
* bc74038 first commit
```

Ce n'est pas ce que l'on veut puisque la branche fusionnée est visible. Dans ce cas, on va commencer par faire un *rebase* sur la branche branch_1. Cette opération va changer la base de la branche pour la

faire démarrer sur le dernier commit de master.

```
$ git log --graph --pretty=oneline --abbrev-commit --all
* 028000a feature #3
| * df0f5ff fixes bug #2
| * d47fbbc fixes bug #1
|/
* b01e2f2 feature #2
* f55ace7 feature #1
* c678075 first commit

$ git rebase master branch_1
Premièrement, rembobinons head pour rejouer votre travail par-dessus...
Application : fixes bug #1
Application : fixes bug #2

$ git checkout master
Basculement sur la branche 'master'
Votre branche est à jour avec 'origin/master'.

$ git merge branch_1
Mise à jour 028000a..4046800
Fast-forward
 fichier1.txt | 2 ++
 1 file changed, 2 insertions(+)

$ git log --graph --pretty=oneline --abbrev-commit
* 4046800 fixes bug #2
* c689cfe fixes bug #1
* 028000a feature #3
* b01e2f2 feature #2
* f55ace7 feature #1
* c678075 first commit
```

Il n'y a pas de bosse ! Il ne reste plus qu'à supprimer la branche branch_1, et le tour est joué.

4.2.2- Je veux que ma branche soit visible

Les deux cas précédents sont toujours les mêmes :

- si la branche cible a bougé depuis la création de la branche source : aucun souci, le comportement par défaut est de faire est un true merge. La branche sera donc visible dans l'historique.
- Si la branche cible n'a pas bougé, pour éviter le fast-forward, il suffit d'ajouter *–no-ff* à la commande *git merge*.

```
$ git merge --no-ff branch_1
```

4.2.3- Dernier cas : git pull

Git pull est la combinaison de deux autres commandes : *git fetch* et *git merge*. En conséquence, lorsque l'on fait un *git pull,* on s'expose au même problème que précédemment, à savoir : faut-il rendre visible le merge ou non ?

En général, la réponse est non.

En effet, le but du git pull est d'intégrer dans sa branche locale les modifications des autres développeurs qui sont présentes sur la branche distante. Il n'y a pas de raison de voir apparaître des merges alors qu'il s'agit en pratique de la même branche.

Pour résoudre ce problème, il suffit de faire (par exemple, pour un pull sur la branche master) :

```
$ git pull --rebase origin master
```

Malheureusement, ce n'est pas complètement suffisant. *Rebase* aura pour effet d'aplatir le graphe de la branche master, mais que faire quand dans ce graphe, il y avait déjà un merge que l'on souhaite garder apparent ? Il faut lancer séparément les commandes :

```
$ git fetch origin master
$ git rebase -p origin/master # -p permet de conserver les merges précédents
```

Une solution plus simple existe depuis la version 1.8.5 de Git :

```
$ git pull --rebase=preserve origin master
```

Afin de le faire systématiquement sans l'oublier, on peut régler ce comportement comme étant celui par défaut de Git :

- Soit dans le dépôt local :

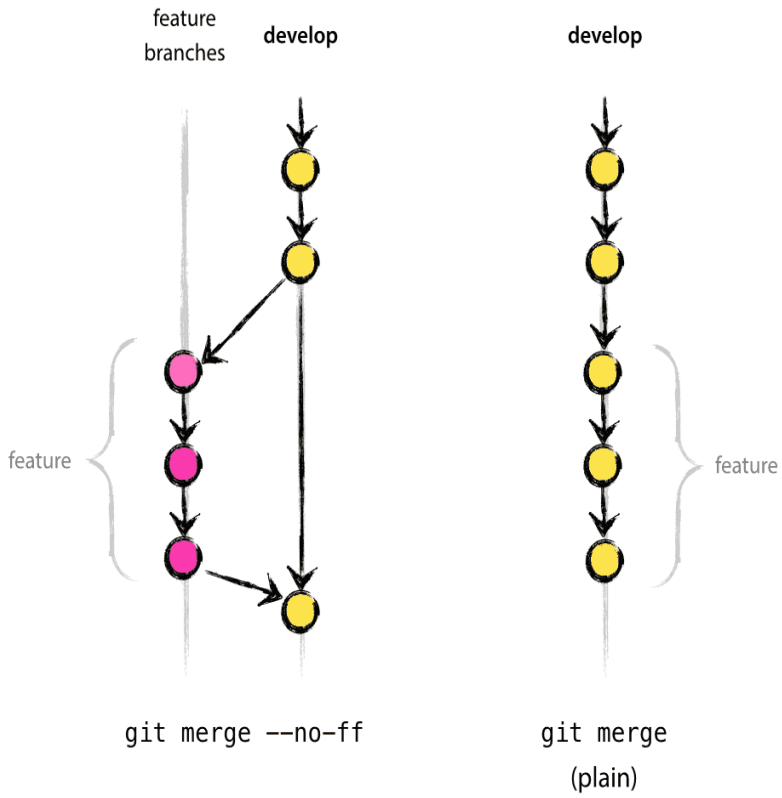
```
$ git config branch.master.rebase preserve
```

- Soit dans la configuration globale :

```
$ git config --global pull.rebase preserve
```

4.2.4- Pour résumer

	Branche source invisible dans l'historique	**Branche source visible dans l'historique**
Branche cible identique	`git merge (fast-forward automatique)`	`git merge --no-ff`
Branche cible changée	`git rebase + git merge`	`git merge (true merge)`



Merge avec et sans fast-forward (cf. [DRI])

Commande	**Effet**
`git merge --no-ff`	Force un true merge
`git rebase <nouvelle base> <branche>`	Déplace l'origine de la branche sur la nouvelle base.
`git rebase -p <nouvelle base> <branche>`	Même chose, mais préserve les anciens true merge.
`git pull --rebase=preserve origin <branche>`	Fait un pull en rebasant les commits locaux et en préservant les anciens true merge.

4.3- Rebase interactif

Le but du rebase interactif est de réécrire l'historique d'une branche. Mais pourquoi vouloir faire ça ? Pour plusieurs raisons :

- Quand on s'y est repris à plusieurs fois pour corriger un bug.
- Quand on s'est égaré d'un sujet à l'autre et que l'on veut réorganiser les commits par blocs

cohérents.

- Quand on a changé d'avis
- Quand on a fait un gros commit qui comprend tout et n'importe quoi (le fameux, avec un message du type « diverses corrections »)
- Quand on a fait une vilaine faute d'orthographe dans son message de commit.

Dans le dernier cas, s'il s'agit du dernier commit, on peut tout de suite faire :

```
$ git commit --amend
```

Pour le reste, on utilisera *git rebase -i*. Imaginons l'historique suivant :

```
$ git log --graph --pretty=oneline --abbrev-commit
* ee44b1b bug #5 corrigé, cette fois c'est la bonne.
* 9a415c9 nouvaile phonctiaunalitet #3
* 2b47cd1 bug #5 corrigé pour de bon
* c5d3b12 correction du bug #5
* 4046800 fixes bug #2
* c689cfe fixes bug #1
* 028000a feature #3
* b01e2f2 feature #2
* f55ace7 feature #1
* c678075 first commit
```

A partir du commit c5d3b12, l'historique est un peu chaotique. Il faudrait regrouper dans un seul commit tout ce qui concerne la correction du bug #5 et corriger l'orthographe déplorable du commit 9a415c9.

```
$ git rebase -i 4046800
```

Cette commande va ouvrir un éditeur de texte, qui affiche :

```
pick c5d3b12 correction du bug #5
pick 2b47cd1 bug #5 corrigé pour de bon
pick 9a415c9 nouvaile phonctiaunalitet #4
pick ee44b1b bug #5 corrigé, cette fois c'est la bonne.

# Rebase 4046800..ee44b1b onto 4046800
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Git nous aide en fournissant le manuel d'utilisation ! Pour chaque commit, on peut donc faire :

- **pick :** on garde le commit tel quel.
- **reword :** on réécrit le message du commit.
- **edit :** on changera complètement un commit (par exemple pour le découper).
- **squash :** on fusionne avec le commit précédent.
- **fixup :** identique à squash, mais utilise le message du commit précédent.
- **exec :** pour ajouter une ligne de commandes
- **supprimer un commit :** en supprimant la ligne correspondante
- **réordonner les commits :** en réordonnant les lignes correspondantes.

Voici donc le fichier proposé :

```
pick c5d3b12 correction du bug #5
fixup 2b47cd1 bug #5 corrigé pour de bon
fixup ee44b1b bug #5 corrigé, cette fois c'est la bonne.
reword 9a415c9 nouvaile phonctiaunalitet #4
```

Il suffit ensuite d'enregistrer le fichier, puis de quitter et git lance le rebase. Évidemment, il peut y avoir quelques soucis en cours de route (le ré-ordonnancement des commits est potentiellement une source de conflits) :

```
$ git rebase -i 4046800
error: impossible d'appliquer ee44b1b... bug #5 corrigé, cette fois c'est la bonne.

Lorsque vous aurez résolu ce problème, lancez "git rebase --continue".
Si vous préférez sauter ce patch, lancez "git rebase --skip" à la place.
Pour extraire la branche d'origine et stopper le rebasage, lancez "git rebase
--abort".

Could not apply ee44b1b3f4c2873239e6399e64691d5b082216ae... bug #5 corrigé, cette
fois c'est la bonne.

$ git status
rebasage en cours ; sur 4046800
Vous êtes en train de rebaser la branche 'master' sur '4046800'.
  (réglez les conflits puis lancez "git rebase --continue")
  (utilisez "git rebase --skip" pour sauter ce patch)
  (utilisez "git rebase --abort" pour extraire la branche d'origine)

Chemins non fusionnés :
  (utilisez "git reset HEAD <fichier>..." pour désindexer)
  (utilisez "git add <fichier>..." pour marquer comme résolu)

      modifié des deux côtés :fichier1.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git
commit -a")
```

Une fois le conflit corrigé, on peut continuer :

```
$ git rebase --continue
```

Au moment d'appliquer les corrections de type *fixup*, voici ce que Git propose :

```
# This is a combination of 3 commits.
# The first commit's message is:
correction du bug #5

# The 2nd commit message will be skipped:

#       bug #5 corrigé pour de bon

# The 3rd commit message will be skipped:

#       bug #5 corrigé, cette fois c'est la bonne.

# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
# rebasage en cours ; sur 4046800
# Vous êtes en train de rebaser la branche 'master' sur '4046800'.
#
# Modifications qui seront validées :
#       modifié:         fichier1.txt
#
```

Le message proposé est celui qui convient, il n'y a qu'à se laisser guider. Arrive ensuite le reword :

```
nouvaile phonctiaunalitet #4

# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
# rebasage en cours ; sur 4046800
# Vous êtes en train de rebaser la branche 'master' sur '4046800'.
#
# Modifications qui seront validées :
#       modifié:         fichier1.txt
#
```

On le corrige, puis on enregistre. Le *rebase* est terminé.

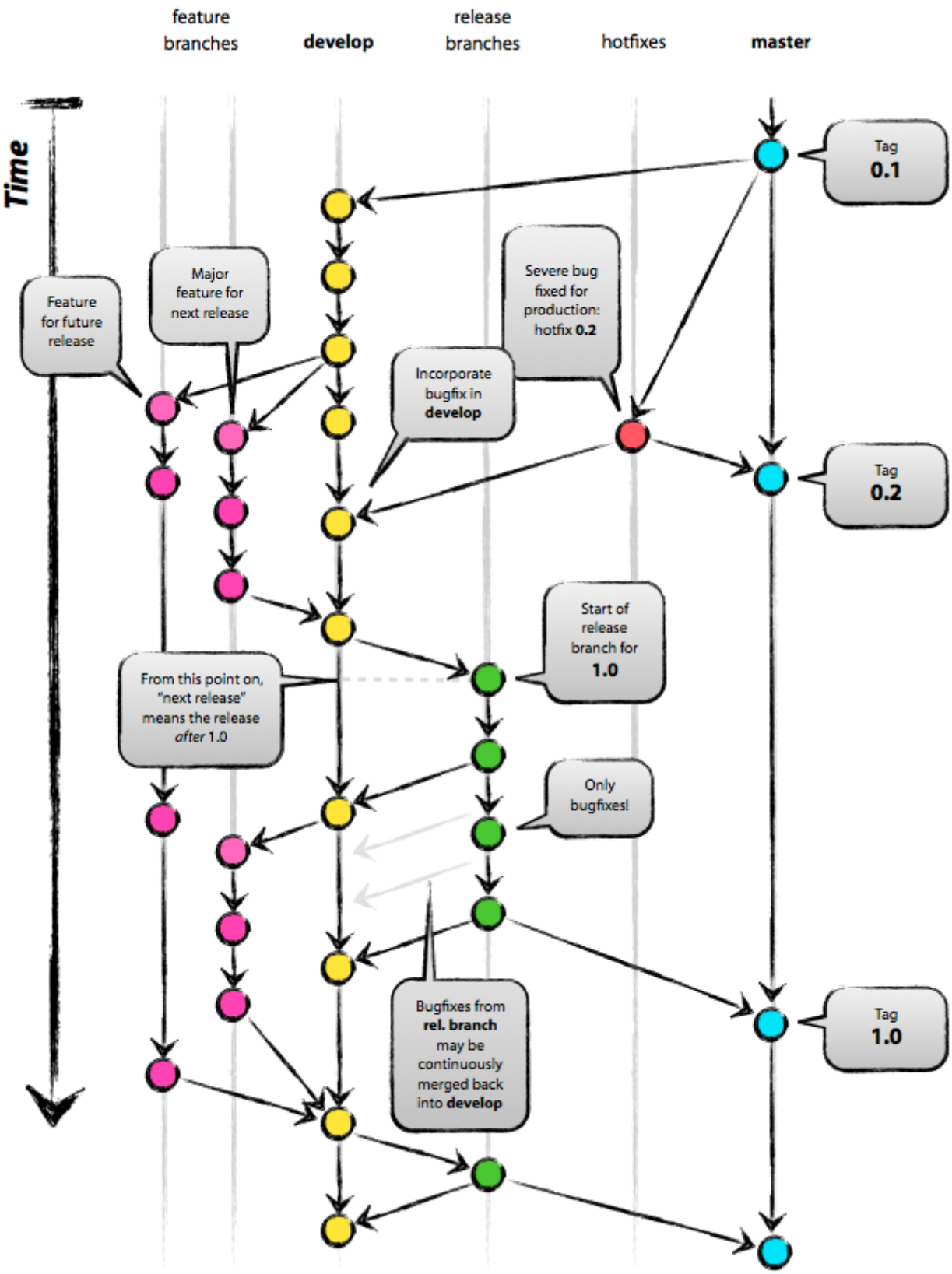
```
[HEAD détachée 9a415c9] feature #4
 1 file changed, 1 deletion(-)
Successfully rebased and updated refs/heads/master.

$ git log --graph --pretty=oneline --abbrev-commit
* 9a415c9 feature #4
* 2ee7974 correction du bug #5
* 4046800 fixes bug #2
* c689cfe fixes bug #1
* 028000a feature #3
* b01e2f2 feature #2
* f55ace7 feature #1
* c678075 first commit
```

Désormais l'historique est propre et on peut faire un *push* pour le partager.

Commande	**Effet**
`git rebase -i <début du rebase>`	Réécrit l'historique de la branche courante.

5- Exemple de gestion des branches sur un projet



Exemple de gestion des branches sur un projet (cf. [DRI])

Ce schéma parle de lui-même. Il représente une gestion idéale des branches sur un projet :

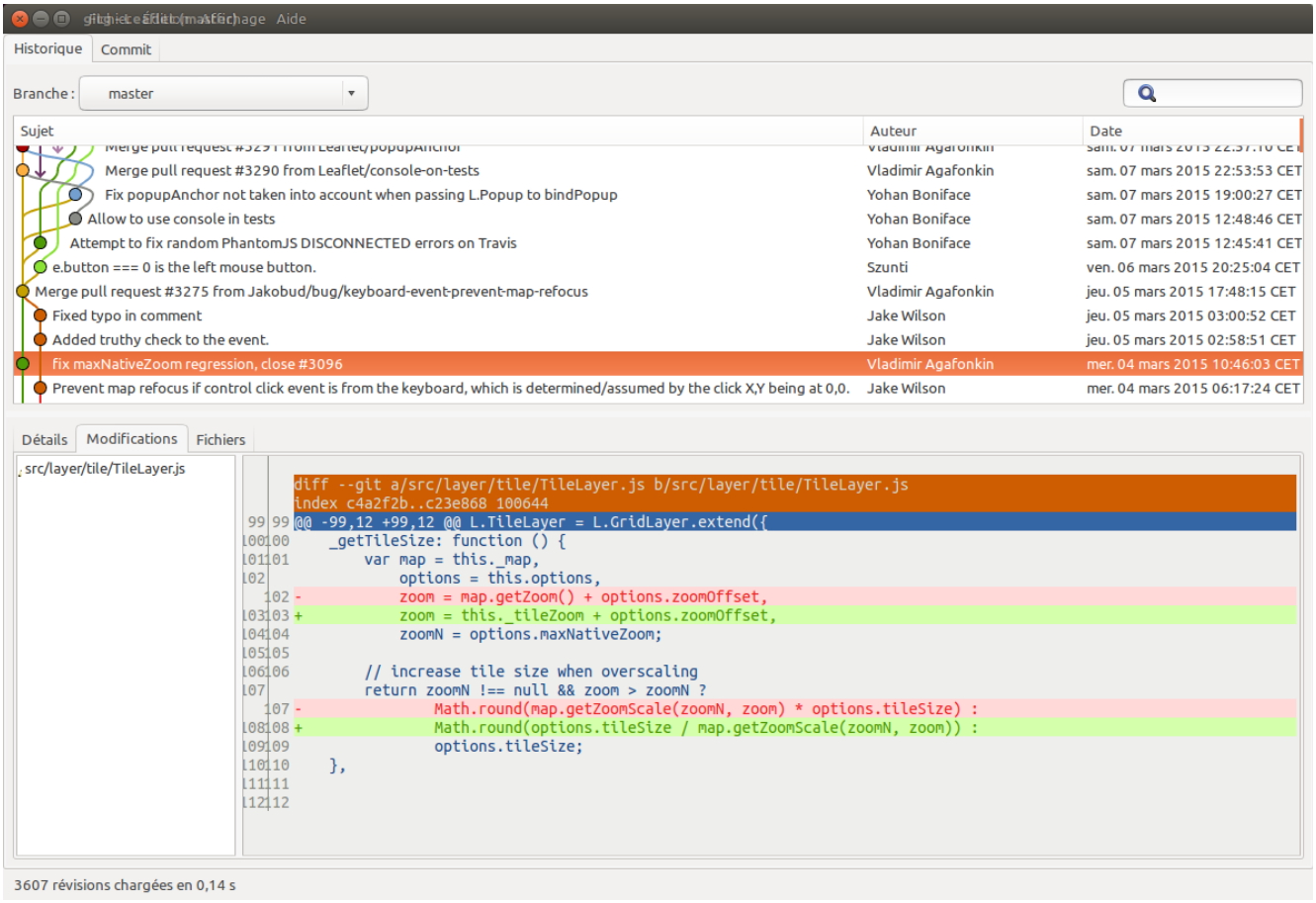
- Des branches de features, dans lesquelles on développe de nouvelles fonctionnalités (une branche par fonctionnalité).
- Lorsque les fonctionnalités sont prêtes, elles sont versées dans une branche de développement qui regroupe toutes les fonctionnalités.
- Lorsque la branche de développement est stable, on crée des branches de release (par exemple 1.0, 1.1, etc). Ces branches serviront à la maintenance des versions en production.
- A chaque fois qu'une opération de maintenance est effectuée, et qu'il faut sortir une nouvelle version du logiciel, le code est versé dans la branche de développement (pour que ces corrections apparaissent dans les futures versions), mais également dans master qui est alors tagué (par exemple 1.0.0, 1.0.1).
- Des branches hotfixes, qui permettent de faire les corrections urgentes.

Il existe un logiciel, git-flow, qui permet de fonctionner de cette façon : http://danielkummer.github.io/git-flow-cheatsheet/. Plutôt que de gérer manuellement ses branches, l'idée est plutôt de déclarer son activité. Par exemple, quand on commence une fonctionnalité, une branche est créée automatiquement, quand on la termine, les merge sont faits dans les bonnes branches, ainsi que les tags.

6- Les clients graphiques

Un rapide coup d'œil sur cette page montre qu'ils sont nombreux : http://git-scm.com/downloads/guis.

On peut gérer intégralement son dépôt avec, ou simplement le visualiser de manière plus attrayante que la ligne de commande.



Dépôt du projet Leaflet (gitg)

Sous Windows, on peut également utiliser TortoiseGit qui est très populaire.

Il convient néanmoins d'être prudent avec ces clients graphiques. Ils ne facilitent pas l'apprentissage de Git, et sont parfois même des pousse-au-crime (options par défaut mal configurées, interfaces peu claires, etc). On a tendance à les utiliser au plus vite sans réfléchir à ce que l'on fait, et les commits sur la mauvaise branche arrivent tout aussi vite (ce n'est qu'un exemple parmi d'autres).

7- Documentation (très) utile

Le guide de référence de Git, tout y est : http://git-scm.com/docs

- Un petit aide-mémoire : https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf
- Un autre aide-mémoire, mais interactif et qui montre parfaitement les interactions entre les cinq zones de Git : http://ndpsoftware.com/git-cheatsheet.html
- Encore un aide-mémoire : http://devdocs.io/git/
- Cours complet sur OpenClassrooms : http://openclassrooms.com/courses/gerez-vos-codes-source-avec-git
- Un résumé très simple et très rapide de l'utilisation de Git au quotidien : http://rogerdudler.github.io/git-guide/
- Bien comprendre l'utilisation du merge et du rebase : http://www.git-attitude.fr/2014/05/04/bien-utiliser-git-merge-et-rebase/

8- Références

[NEB] M.Nebra, http://openclassrooms.com/courses/gerez-vos-codes-source-avec-git

[GITREF] http://git-scm.com/doc

[DRI] V. Driessen, http://nvie.com/posts/a-successful-git-branching-model/